Matt's Code

Matt James

Contents

Chapter 1

Introduction

This document lists a bunch of the GitHub repositories created by me which may be useful to others. Some of these repositories are fairly complete, others are less so. I will do my best to fix and update anything that is buggy or incomplete, please do report bugs in the relevant repositories if you can. If you're feeling particularly helpful - feel free to send pull requests.

Most of the code here is written in Python, some things make use of C++ libraries to do some of the heavy lifting, one is a pretty dodgy Python wrapper of a C++ wrapper of Fortran code... Some of the modules and libraries used here are dependencies of others. In the more complete repos `pip` will take care of dependencies, otherwise some manual installation may be required.

1.1 Setting up the environment

In this section I describe how to set up the environment such that everything *should* pretty much work...

1.1.1 Linux

If running on ALICE/SPECTRE, you will most likely be required to enable the following modules:

```
module load gcc/9.3
module load python/gcc/3.9.10
module load git/2.35.2
```

where exact version numbers may change (use whatever is latest, don't just copy and paste!) and the replacement for SPECTRE/ALICE may have another method for loading these things in for all I know. I also recommend adding those to the end of your `/.bashrc` file so that they load every login, e.g.:

```
echo module load gcc/9.3 >> ~/.bashrc
echo module load python/gcc/3.9.10 >> ~/.bashrc
echo module load git/2.35.2 >> ~/.bashrc
```

The above is unlikely to be necessary on a local Linux installation, instead I would recommend installing `git, gcc, g++, make, gfortran` and `pip3`, e.g. in Ubuntu:

```
sudo apt install git gcc g++ binutils gfortran python3-pip
```

All of the above should allow you to install/run/compile most of my code. I wouldn't recommend using Conda in Linux - I know it has cause some problems/confusion when it comes to linking Python with C/C++ on SPECTRE.

1.1.2 Windows

A fair portion of the code is able to run on Windows - much of the Python code is platform independent and some of the C++ libraries/backends are able to be compiled using Windows. In this case, I *would* actually recommend installing Conda, as it worked for me. The GCC compilers (for C/C++/Fortran) can all be installed easily with TDM-GCC (get the 64-bit version here), just remember to put a tick in the box for "fortran" and "openmp".

1.1.3 MacOS

I managed to install the relevant packages in a virtual Hackintosh once. I don't remember how, perhaps using homebrew. Good luck...

1.2 Setting up a virtual environment

In SPECTRE I never actually bothered with a virtual environment, mistakes were made, headaches may have been avoided had I done so. This step is entirely optional, but somewhat recommended:

```
#create a virtual environment, call it what you want,
#here I call mine "env"
python3 -m venv env
```

Once this has been created, you MUST activate it before running any code, or you will just be running things globally:

```
source env/bin/activate
```

note that I am assuming that `env` exists in the current working directory, if not adjust the path accordingly! If it works, the prompt terminal prmpt should change, e.g:

```
#before:
matt@matt-MS-7B86:~$

source env/bin/activate
#after:
(env) matt@matt-MS-7B86:~$
```

1.3 Some python packages

Here are a list of Python packages which are either going to be required by most of my code, or would just be recommended:

1. ipython : best Python interpreter, forget notebooks
2. numpy : essential, don't skip
3. matplotlib : for plotting
4. scipy : loads of good stuff here
5. wheel : used to build Python packages to be installed by pip
6. cdflib : reads CDF files
7. keras : nice for machine learning
8. tensorflow : also machine learning

install them:

```
#update pip first
python3 -m install pip --upgrade --user

pip3 install ipython numpy matplotlib scipy wheel cdflib keras tensorflow --user
```

where the "`--user`" flag may or may not be necessary, depending on your version of Python - it places the installed modules in `~/.local/lib/python3.9/site-packages`.

In theory, at this point you should be able to run `ipython3` (or just `ipython`) within the terminal, from which any installed code can be imported. The reason I recommend using Ipython over the standard Python interpreter is that it has autocomplete and it uses pretty colours for syntax highlighting. It would also be a good idea to enable the autoreload feature in Ipython, which recompiles anything that has been edited since it was last run, otherwise would have to reload the code manually (or restart the session) after every edit. Run

```
ipython profile create
```

then add the following lines to ∼/.ipython/profile_default/ipython_config.py:

```
c.InteractiveShellApp.extensions = ['autoreload']
c.InteractiveShellApp.exec_lines = ['%autoreload 2']
c.InteractiveShellApp.exec_lines.append('print("Warning: disable autoreload in ipython_config.py to imp
```

That should just about do it.

Chapter 2

Plasma Models

2.1 `spicedmodel`: The Scalable Plasma Ion Composition and Electron Density Model

Python wrapper for the Scalable Plasma Ion Composition and Electron Density (SPICED) model:

James, M. K., Yeoman, T.K., Jones, P., Sandhu, J. K., Goldstein, J. (2021), The Scalable Plasma Ion Composition and Electron Density (SPICED) model for Earth's inner magnetosphere, *J. Geophys. Res. Space Physics*, https://doi.org/10.1029/2021JA029565

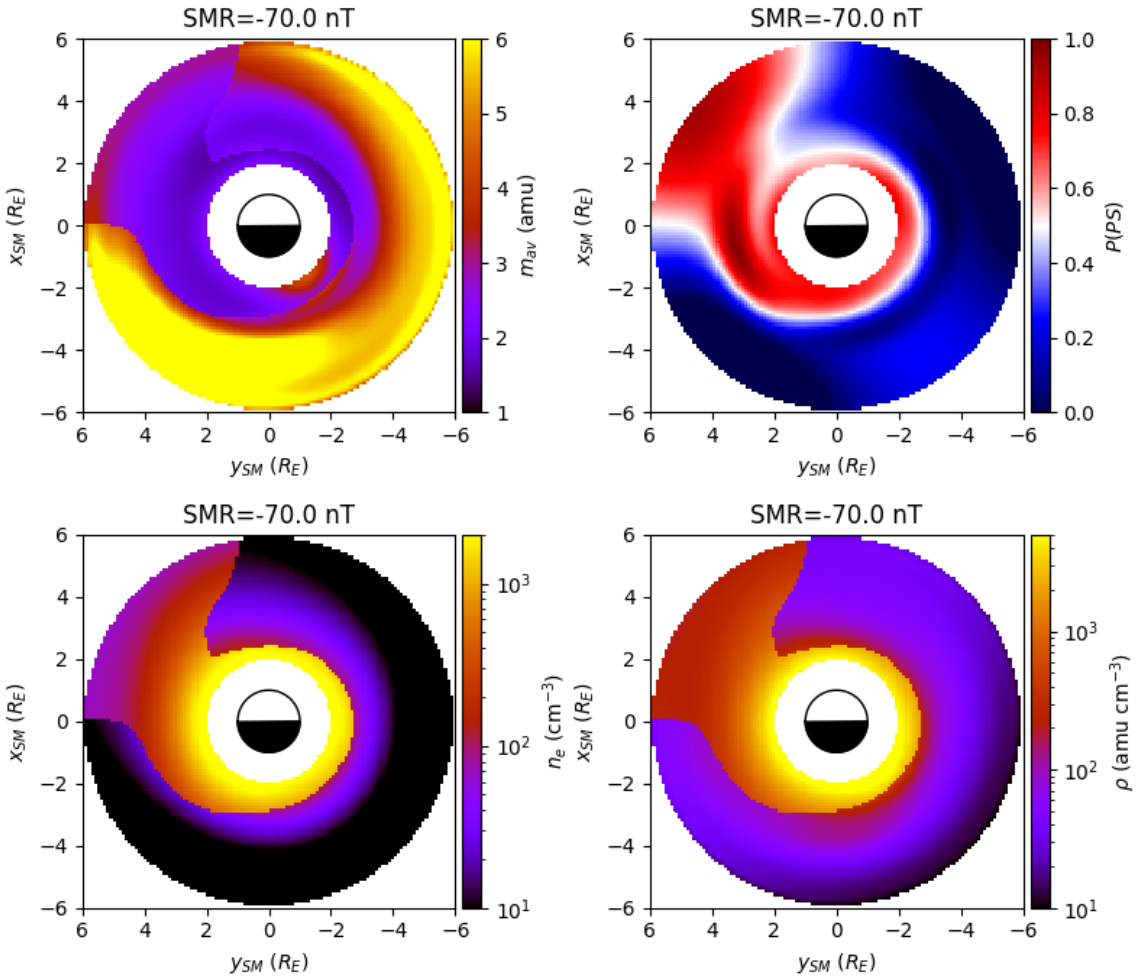

Figure 2.1: Example output of the SPICED model.

2.1.1 Installation

Using `pip`

This will download the package from PyPI:

```
pip3 install spicedmodel --user
```

From Source

Obtain the latest release from https://github.com/mattkjames7/spicedmodel

```
git clone https://github.com/mattkjames7/spicedmodel
cd spicedmodel
```

Either install using `setup.py`:

```
python3 setup.py install --user
```

or by building a wheel:

```
python3 setup.py bdist_wheel
pip3 install dist/spicedmodel-XXX.whl --user
```

where `"XXX"` is the rest of the file name, which will vary depending upon the current version.

2.1.2 Usage

Load `python3` or `ipython3`, and import

```
import spicedmodel
```

Accessing the models

There are four models, plus two additional combinations of these models:

- Plasmasphere average ion mass, $m_{av,ps}$: `spicedmodel.MavPS`
- Plasmatrough average ion mass, $m_{av,pt}$: `spicedmodel.MavPT`
- Combined average ion mass, m_{av}: `spicedmodel.Mav`
- Hot average ion mass, m_{av}: `spicedmodel.MavHot`
- Probability of being within the plasmasphere, P: `spicedmodel.Prob`
- Plasmasphere electron density, $n_{e,ps}$: `spicedmodel.PS`
- Plasmatrough electron density, $n_{e,pt}$: `spicedmodel.PT`
- Combined electron density, n_e (a combination of plasmasphere, plasmatrough and probability models): `spicedmodel.Density`
- Combined plasma mass density, ρ: `spicedmodel.PMD`

The average versions of each model can be accessed simply by providing the positions in the equatorial plane where you would like them, e.g.:

```
#either using SM x and y coordinates
P = spicedmodel.Prob(x,y)

#or using MLT (M) and L-Shell (L)
P = spicedmodel.Prob(M,L,Coord='ml')
```

The scaled models can be accessed using the same functions, this time including the `SMR` keyword (for `Mav`, `MavPS`, `MavPT`, `Prob`, `PS`, `PT`, `Density` or `PMD`) or `F107` (for `MavHot`), e.g.:

```
#electron density
ne = spicedmodel.Density(x,y,SMR=-75.0)

#average ion mass
mav = spicedmodel.Mav(x,y,SMR=-75.0)

#plasma mass density, effectively ne*mav
pmd = spicedmodel.PMD(x,y,SMR=-75.0)
```

Plotting the models

A simple function is included, `PlotEq`, which allows the plotting of any of the models in the equatorial plane, e.g.:

```
ax = spicedmodel.PlotEq(ptype,SMR=-75.0)
```

where `ptype` is used to tell the function which model to plot, available options are: `'mav'|'mavps'|'mavpt'|'mavhot'|'p`
The following code produces a plot with all 6 models when SMR = -75 nT

```
import matplotlib.pyplot as plt
import spicedmodel

#create the plot window
plt.figure(figsize=(8,7))

#set the parameters of the models
smr = -70.0

#plot the average ion mass
ax0 = spicedmodel.PlotEq('mav',SMR=smr,fig=plt,maps=[2,2,0,0])

#plot probability
ax1 = spicedmodel.PlotEq('prob',SMR=smr,fig=plt,maps=[2,2,1,0])

#plot electron density
ax4 = spicedmodel.PlotEq('density',SMR=smr,fig=plt,maps=[2,2,0,1])

#plot plasma mass density
ax5 = spicedmodel.PlotEq('pmd',SMR=smr,fig=plt,maps=[2,2,1,1])

#adjust everything to fit
plt.tight_layout()
```

2.2 spiced

GitHub: https://github.com/mattkjames7/spiced.git

The C++ code behind the SPICED model. It should be possible to build this library in Linux, Windows and MacOS.

2.2.1 Installation

In Linux and MacOS, it should eb possible to make and install the library:

```
make

#optionally install globally
sudo make install
```

Or in Windows:

```
compile.bat
```

2.2.2 Usage

When using this library, the header file should be included, i.e.:

```
#include <spiced.h>
```

and the linker flag `-lspiced` should be used during compilation.

The models also need to be initialized at runtime using `initModels();`. `spiced.h` contains a full list of the functions which can be linked to using C and other languages like Python within the `extern "C" {}` section; other symbols outside this section can, such as the model objects themselves may be interacted with directly using C++.

2.3 `HermeanFLRModel`: Model of Mercury's dayside plasma mass density

GitHub: https://github.com/mattkjames7/HermeanFLRModel.git

Estimate the dayside plasma mass density in Mercury's magnetosphere using the field line resonance (FLR) based model from James2019.

2.3.1 Installation

This hasn't been placed on PyPI, so either download fromt he GitHub page and install using `pip`, or clone and build the package:

```
#if you download the package
pip3 install HermeanFLRModel-x.y.z-py3-none-any.whl --user

#or clone, build and install
git clone https://github.com/mattkjames7/HermeanFLRModel.git
cd HermeanFLRModel
python3 setup.py bdist_wheel
pip3 install dist/HermeanFLRModel-x.y.z-py3-none-any.whl --user
```

where `x.y.z` should be replaced with the current version number.

2.3.2 Usage

Import the module and create an instance of the `Model` object:

```
import HermeanFLRModel as hflr

model = hflr.Model(Alpha,Coord='MSM')
```

where `Alpha` is the power law index which should be an integer from 0 to 6, and `Coord` sets the coordinate system to use (either MSM or MSO).

Use the `model.Calc()` member function to work out densities:

```
#create input coordinate(s)
x = np.zeros(6)
y = np.array([1.0,1.2,1.4,1.6,1.8,2.0])
z = np.zeros(6)

#call model Calc() function
rho = model.Calc(x,y,z)
```

Or produce a plot of the plasma mass denity for a slice through the magnetosphere, e.g.:

```
#select magnetic local time and alpha
MLT = 6.0
Alpha = 3.0

#plot it
ax = hflr.PlotModelSlice(MLT,Alpha)
```

which should produce something like figure 2.2.

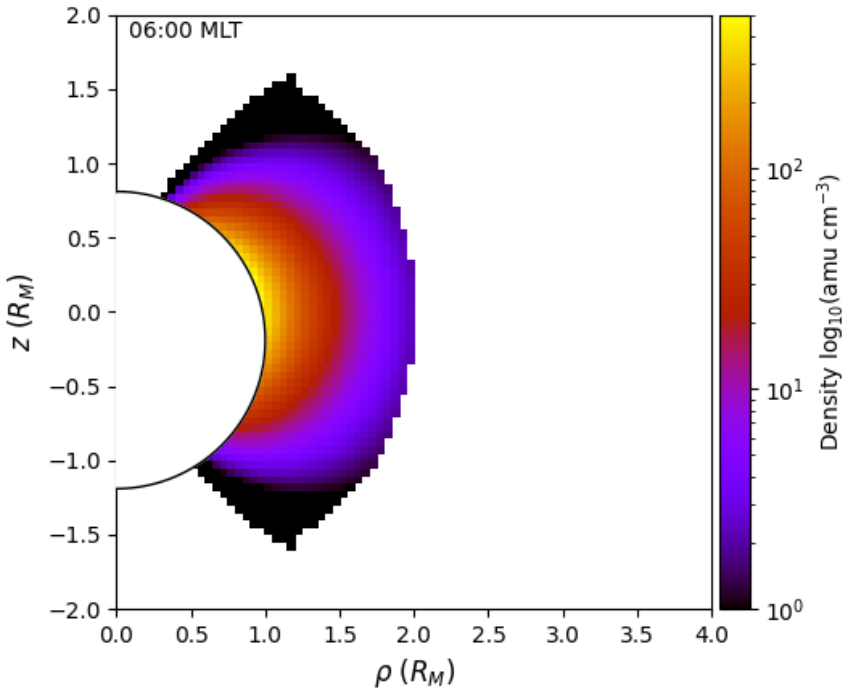

Figure 2.2: Plasma mass density at 6:00 MLT.

2.4 PyGCPM: Wrapper for the Global Core Plasma Model

GitHub: https://github.com/mattkjames7/PyGCPM.git

This is a Python wrapper for the Global Core Plasma Model (Gallagher2000, code found here)

2.4.1 Installation

This module exists on PyPI, so can be installed using `pip`:

```
pip3 install PyGCPM --user
```

2.4.2 Usage

There are three functions:

1. `PyGCPM.GCPM()`: provides particle densities at positions defined in SM coordinates.
2. `PyGCPM.PlotEqSlice()` : plots the density of a species in the equatorial plane.
3. `PyGCPM.PlotMLTSlice()` : plots the density of a particle species in

Firstly, get some densities at some positions in SM coordinates, units of R_E:

```
import PyGCPM
ne,nH,nHe,nO = PyGCPM.GCPM(x,y,z,Date,ut,Kp=Kp,Verbose=Verbose)
```

where `Date` is the date in the format yyyymmdd, `ut` is the time in hours, `Kp` is the Kp index and `Verbose=True` would display progress. The outputs of this function `ne`, `nH`, `nHe` and `nO` are the densities of electrons, protons, helium ions and oxygen ions, respectively.

We can plot the density of a particle species in the equatorial plane:

```
PyGCPM.PlotEqSlice(20010902,12.0,Parameter='ne')
```

which should produce the plot in figure 2.3.

We can also plot the density of a particle species in a slice of MLT:

```
PyGCPM.PlotMLTSlice(8.0,20010902,12.0,Parameter='ne')
```

which should produce the plot in figure 2.4.

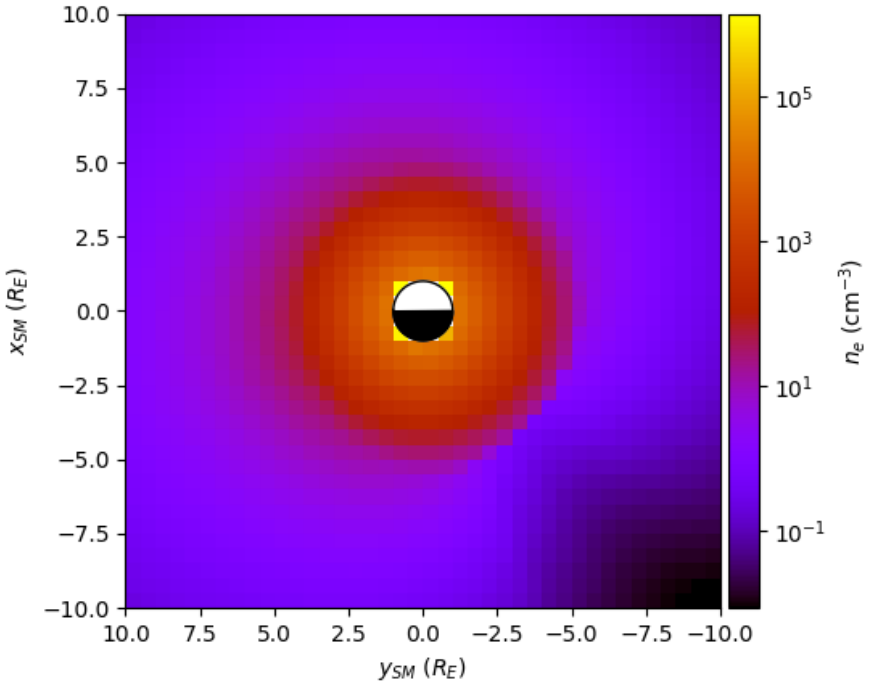

Figure 2.3: Equatorial electon density.

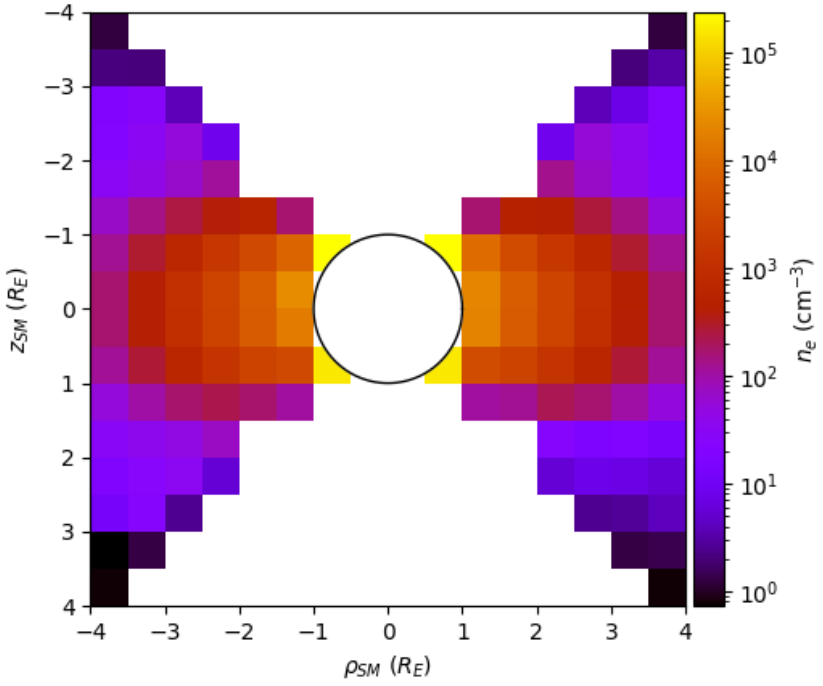

Figure 2.4: MLT slice of electon density.

Chapter 3

Field Models

3.1 `PyGeopack`: Python wrapper for the Tsyganenko field models

This is a Python module for obtaining field vectors and traces of the Tsygenenko field models. It is a wrapper of a wrapper (see 3.2). The latest code can be viewed and downloaded from here: https://github.com/mattkjames7/PyGeopack.

3.1.1 Installation

The easiest way to install `PyGeopack` is using `pip`, e.g.:

```
pip3 install PyGeopack --user
```

for other installation methods, see the GitHub repo.

At this point, it *may* just work if you were to try to import it, but there's a reasonably good chance that the C++/Fortran code will need to be recompiled, in which case we need to ensure that there are compilers available to do this (see section sectSetup).

One of the features of PyGeopack is that it can easily package together all of the geomagnetic/solar wind parameters that the models use so that when you request a trace or a field vector for a specific date and time, it will automatically try to find the appropriate parameters. This isn't strictly necessary for the models to work, as they will default to some fairly average parameters and they can be overridden manually. In order to be able to use this functionality, this module and the submodules which it relies on to collect the relevant data need to know where they can store the parameters. This means exporting a few environment variables (e.g. in ~`/.bashrc`):

```
export KPDATA_PATH=/path/to/kp
export OMNIDATA_PATH=/path/to/omni
export GEOPACK_PATH=/path/to/geopack/data
```

which are set as follows for me on SPECTRE:

```
export KPDATA_PATH="/data/sol-ionosphere/mkj13/Kp"
export OMNIDATA_PATH="/data/sol-ionosphere/mkj13/OMNI"
export GEOPACK_PATH="/data/sol-ionosphere/mkj13/Geopack"
```

Once that is done, it should work...

3.1.2 Usage

The first time this is imported, there is a good chance that it will attempt to recompile itself. There will be a lot of messages on the screen, but it should finish successfully. If it fails, double check that you have the required compilers installed, raise an issue on the GitHub page if the problem persists.

If you would lke the latest model parameters, run the following:

```
import PyGeopack as gp
gp.Params.UpdateParameters(SkipWParameters=True)
```

This may take a little time, depending on how much data it needs to download. It will take all of the data and compile it into one binary file ~350 MiB in size, once this is done, it should be relatively quick loading the data into memory.

The model field vectors can be returned using the `ModelField` function:

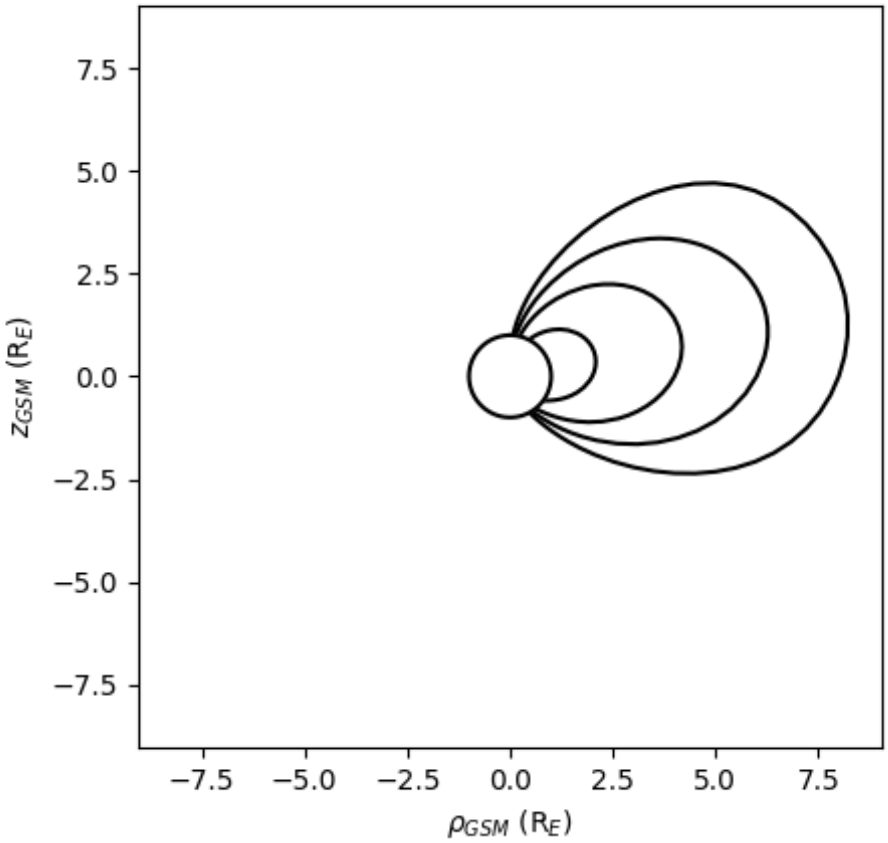

Figure 3.1: Example of field tracing using `PyGeopack`.

```
Bx,By,Bz = gp.ModelField(x,y,z,Date,ut,Model='T96',CoordIn='GSM',**kwargs)
```

where `x`, `y` and `z` can be scalars or arrays of position, in units of Earth radii and in the coordinate system defined by the `CoordIn` keyword (`'GSE'|'GSM'|'SM'`). `Date` can be an array or scalar of date(s) in the format yyyymmdd, while `ut` is in hours from the start of the day. The models currently available are `'T89'`, `'T96'`, `'T01'` and `'TS05'`.

Traces are simple to produce and can be done a single trace at a time, or in batches, e.g.:

```
import numpy as np

#define a few starting positions for the traces
x = np.array([2.0,4.0,6.0,8.0])
y = np.array([0.0,0.0,0.0,0.0])
z = np.array([0.0,0.0,0.0,0.0])

#run the traces, return TraceField object
T = gp.TraceField(x,y,z,20221222,16.0)

#plot field traces
ax = T.PlotRhoZ()
```

which should produce a plot similar to figure 3.1.

For more information on the keyword arguments please see the readme.

3.2 geopack: C++ wrapper for Tsyganenko field models

This is the code that `PyGeopack` calls, it provides a simple C-compatible interface for calculating field vectors, tracing and coordinate conversion. The C++ code in this library is used for configuring model parameters,

determining footprints and providing a simple interface for the models, while the Fortran code currently provides field vectors, coordinate transforms and tracing.

3.2.1 Installation

This code can be installed as a linkable library (at least on POSIX systems):

```
#clone the repo
git clone https://github.com/mattkjames7/geopack --recurse-submodules

#cd into it
cd geopack

#fetch the submodules if you forgot the
#--recurse-submodules flag on the clone command
git submodule update --init --recursive

#compile it
make
sudo make install
```

On Linux and MacOS the `sudo make install` command will place the header file at `/usr/local/include/geopack.h` and the shared object file at `/usr/local/lib/libgeopack.so`, unless the `PREFIX` keyword is set (by default `PREFIX=/usr/local`). If the `PREFIX` uses a custom path, then be sure to let the linker know where it exists, either by setting `LD_LIBRARY_PATH` or by using `-L` and `-I`.

On Windows, run `compile.bat` to create `lib
libgeopack.dll`.

3.2.2 Usage

To use this code with C and C++, the header file must be included, i.e.:

```
#include <geopack.h>
```

and when compiling, the `-lgeopack` flag should be used to link to the library.

C and other languages such as Python should use the wrapper functions defined within the `extern "C" {}` section of `geopack.h`. This includes `ModelField()` for calculating model field vectors, `TraceField()` for field traces and a number of functions for coordinate conversion, e.g. `SMtoGSMUT()`. An example of how to trace a field line is in `geopack.c`.

C++ is able to use all of the functions declared in `geopack.h`, including the wrapper functions used by C. This means that direct usage of the `Trace` class is possible, an example of this is shown in `geopack.cc`.

3.3 `libinternalfield`: C++ spherical harmonic model code

This library provides field vectors for spherical harmonic magnetic field models. It has a bunch of built in models from magnetized planets and a moon (Ganymede). This library is used by libjupitermag.

3.3.1 Installation

Compile and install in Linux and MacOS:

```
#clone the repo
git clone https://github.com/mattkjames7/libinternalfield

#cd into it
cd libjupitermag

#compile it
make
sudo make install
```

Or in Windows, run `compile.bat` to create `libinternalfield.dll`.

3.3.2 Usage

To use this library, the header should be included `include <internalfield.h>` and the code should be compiled with the `-linternalfield` flag in order to link to the library.

In C, we can use the `getModelFieldPointer()` to get a function pointer to the model we want to use, e.g.:

```c
#include <stdio.h>
#include <internalfield.h>

int main() {

        printf("Testing C\n");

        /* try getting a model function */
        modelFieldPtr model = getModelFieldPtr("jrm33");
        double x = 10.0;
        double y = 10.0;
        double z = 0.0;
        double Bx, By, Bz;
        model(x,y,z,&Bx,&By,&Bz);

        printf("B = [%6.1f,%6.1f,%6.1f] nT at [%4.1f,%4.1f,%4.1f]\n",Bx,By,Bz,x,y,z);

        printf("C test done\n");

}
```

C++ can use the `internalModel` object, e.g.:

```c
#include <internal.h>

int main() {
    /* set current model */
    internalModel.SetModel("jrm09");

    /* set intput and output coordinates to Cartesian */
    internalModel.SetCartIn(true);
    internalModel.SetCartOut(true);

    /* input position (cartesian)*/
    double x = 35.0;
    double y = 10.0;
    double z = -4.0;

    /* output field */
    double Bx, By, Bz;
    internalModel.Field(x,y,z,&Bx,&By,&Bz);
}
```

3.4 `vsmodel`: Python based Volland-Stern electric field model for Earth

This is a purely Python-based implementation of the Volland-Stern electric field model Volland1973,Stern1975.

3.4.1 Installation

Use `pip`:

```
pip3 install vsmodel --user
```

3.4.2 Usage

Electric field vectors can be determined using either cylindrical (`vsmodel.ModelE`) or Cartesian (`vsmodel.ModelCart`) coordinates (Solar Magnetic), e.g.:

```
import vsmodel

##### The simple model using Maynard and Chen ####
#the Cartesian model
Ex,Ey,Ez = vsmodel.ModelCart(x,y,Kp)

#the cylindrical model
Er,Ep,Ez = vsmodel.ModelE(r,phi,Kp)


#### The Goldstein et al 2005 version ####
#the Cartesian model, either by providing solar wind
#speed (Vsw) and IMF Bz (Bz), or the equivalent E field (Esw)
Ex,Ey,Ez = vsmodel.ModelCart(x,y,Kp,Vsw=Vsw,Bz=Bz)
Ex,Ey,Ez = vsmodel.ModelCart(x,y,Kp,Esw=Esw)

#the cylindrical model
Er,Ep,Ez = vsmodel.ModelE(r,phi,Kp,Vsw=Vsw,Bz=Bz)
Er,Ep,Ez = vsmodel.ModelE(r,phi,Kp,Esw=Esw)
```

where the Maynard1975 method uses the Kp index and the Goldstein2005 method uses Kp, solar wind speed and the z-component of the interplanetary magnetic field.

The electric field magnitude and $\mathbf{E} \times \mathbf{B}$ velocity can be plotted in the Earth's equatorial plane:

```
import vsmodel
import matplotlib.pyplot as plt

plt.figure(figsize=(9,8))
ax0 = vsmodel.PlotModelEq('E',Kp=1.0,Vsw=-400.0,Bz=-2.5,
                maps=[2,2,0,0],fig=plt,fmt='%4.2f',scale=[0.01,10.0])
ax1 = vsmodel.PlotModelEq('E',Kp=5.0,Vsw=-400.0,Bz=-2.5,
                maps=[2,2,1,0],fig=plt,fmt='%4.2f',scale=[0.01,10.0])
ax2 = vsmodel.PlotModelEq('V',Kp=1.0,Vsw=-400.0,Bz=-2.5,
                maps=[2,2,0,1],fig=plt,scale=[100.0,10000.0])
ax3 = vsmodel.PlotModelEq('V',Kp=5.0,Vsw=-400.0,Bz=-2.5,
                maps=[2,2,1,1],fig=plt,scale=[100.0,10000.0])
ax0.set_title('$K_p=1$; $E_{sw}=-1$ mV m$^{-1}$')
ax2.set_title('$K_p=1$; $E_{sw}=-1$ mV m$^{-1}$')
ax3.set_title('$K_p=5$; $E_{sw}=-1$ mV m$^{-1}$')
ax1.set_title('$K_p=5$; $E_{sw}=-1$ mV m$^{-1}$')
plt.tight_layout()
```

which would produce figure 3.2

3.5 JupiterMag: Python wrapper for Jovian field models

GitHub: https://github.com/mattkjames7/JupiterMag.git

Python wrapper for a collection of Jovian magnetic field models written in C++ (see libjupitermag).

This is part of a community code project : Magnetospheres of the Outer Planets Group Community Code

3.5.1 Requirements

For the Python code to run (without rebuilding the C++ backend), the following Python packages would be required:

- NumPy
- Matplotlib

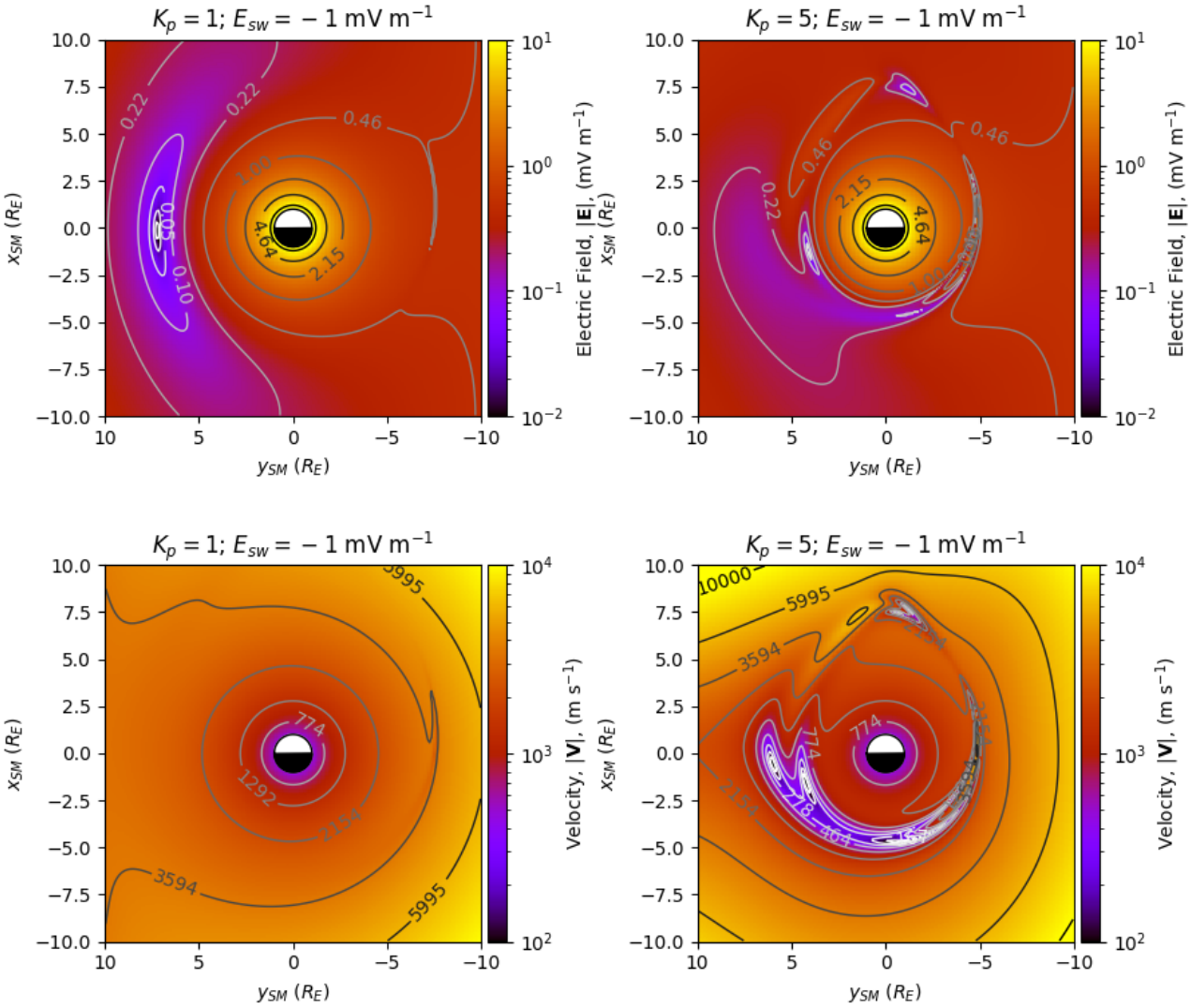

Figure 3.2: Volland-Stern electric field (top plots) and $\mathbf{E} \times \mathbf{B}$ velocity (bottom plots).

- DateTimeTools
- RecarrayTools
- PyFileIO

All of which would be installed automatically if using `pip`.

On some systems, the shared object files would need rebuilding before they can be loaded and accessed using Python. Upon the first import of the `JupiterMag` module, if the shared object/DLL fails to load then it will attempt to use a local C++ compiler to rebuild the binaries.

Linux

JupiterMag was built and tested primarily using Linux Mint 20.3 (based on Ubuntu 20.04/Debian). To rebuild the code, ensure that `g++`, `make`, and `ld` are installed.

Windows

This has been tested on Windows 10 (64-bit), other versions may also work. Requires `g++` and `ld` to work (these can be provided by TDM-GCC). This may or may not work with other compilers installed.

MacOS

This module has been tested on MacOS 11 Big Sur. It requires `g++`, `make`, and `libtool` to recompile (provided by Xcode).

3.5.2 Installation

Install using `pip3`:

```
pip3 install JupiterMag --user
```

Download the latest release (on the right → if you're viewing this on GitHub), then from within the directory where it was saved:

```
pip3 install JupiterMag-1.0.0-py3-none-any.whl --user
```

Or using this repo (replace "1.0.0" with the current version number):

```
#pull this repo
git clone https://github.com/mattkjames7/JupiterMag.git
cd JupiterMag

#update the submodule
git submodule update --init --recursive

#build the wheel file
python3 setup.py bdist_wheel
#the output of the previous command should give some indication of
#the current version number. If it's not obvious then do
# $ls dist/ to see what the latest version is
pip3 install dist/JupiterMag-1.0.0-py3-none-any.whl --user
```

I recommend installing `gcc` $\geq$ 9.3 (that's what this is tested with, earlier versions may not support the required features of C++).

This module should now work with both Windows and MacOS.

Update an Existing Installation

To update an existing installation:

```
pip3 install JupiterMag --upgrade --user
```

Alternatively, uninstall then reinstall, e.g.:

```
pip3 uninstall JupiterMag
pip3 install JupiterMag --user
```

3.5.3 Usage

Internal Field

A number of internal field models are included (see here for more information) and can be accessed via the `JupiterMag.Internal` submodule, e.g.:

```
import JupiterMag as jm

#configure model to use VIP4 in polar coords (r,t,p)
jm.Internal.Config(Model="vip4",CartesianIn=False,CartesianOut=False)
Br,Bt,Bp = jm.Internal.Field(r,t,p)

#or use jrm33 in cartesian coordinates (x,y,z)
jm.Internal.Config(Model="jrm33",CartesianIn=True,CartesianOut=True)
Bx,By,Bz = jm.Internal.Field(x,y,z)
```

All coordinates are either in planetary radii (x, y, z, r) or radians (t, p). All Jovian models here use $R_j = 71,492$ km.

External Field

Currently, the only external field source included is the Con2020 field (see here for the standalone Python code and here for more information on the C++ code used here as part of libjupitermag), other models could be added in the future.

This works in a similar way to the internal field, e.g.:

```
#configure model
jm.Con2020.Config(equation_type='analytic')
Bx,By,Bz = jm.Con2020.Field(x,y,z)
```

Tracing

Field line tracing can be done using the `TraceField` object, e.g.

```
import JupiterMag as jm

#configure external field model prior to tracing
#in this case using the analytic Con2020 model for speed
jm.Con2020.Config(equation_type='analytic')

#trace the field in both directions from a starting position
T = jm.TraceField(5.0,0.0,0.0,IntModel='jrm09',ExtModel='Con2020')
```

The above example will trace the field line from the Cartesian SIII position (5.0,0.0,0.0) (R_j) in both directions until it reaches the planet using the JRM09 internal field model with the Con2020 external field model. The object returned, `T`, is an instance of the `TraceField` class which contains the positions and magnetic field vectors at each step along the trace, along with some footprint coordinates and member functions which can be used for plotting.

A longer example below can be used to compare field traces using just an internal field model (JRM33) with both internal and external field models (JRM33 + Con2020):

```
import JupiterMag as jm
import numpy as np

#be sure to configure external field model prior to tracing
jm.Con2020.Config(equation_type='analytic')
#this may also become necessary with internal models in the future, e.g.
#setting the model degree

#create some starting positions
n = 8
theta = (180.0 - np.linspace(22.5,35,n))*np.pi/180.0
r = np.ones(n)
```

```
x0 = r*np.sin(theta)
y0 = np.zeros(n)
z0 = r*np.cos(theta)

#create trace objects, pass starting position(s) x0,y0,z0
T0 = jm.TraceField(x0,y0,z0,Verbose=True,IntModel='jrm33',ExtModel='none')
T1 = jm.TraceField(x0,y0,z0,Verbose=True,IntModel='jrm33',ExtModel='Con2020')

#plot a trace
ax = T0.PlotRhoZ(label='JRM33',color='black')
ax = T1.PlotRhoZ(fig=ax,label='JRM33 + Con2020',color='red')

ax.set_xlim(-2.0,15.0)
ax.set_ylim(-6.0,6.0)
```

The resulting objects T0 and T1 store arrays of trace positions and magnetic field vectors along with a bunch of footprints. The above code produces figure 3.3.

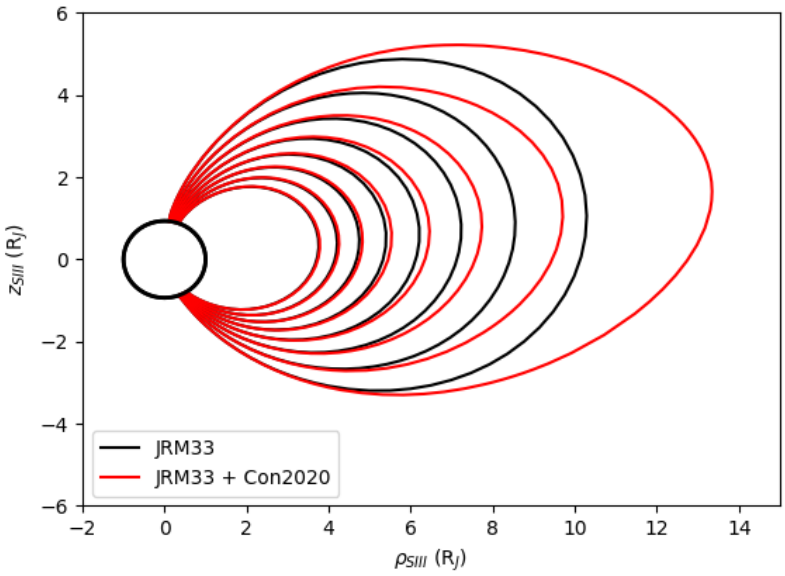

Figure 3.3: Comparison between traces using only the internal field (black) and traces also using a magnetodisc model (red).

3.6 libjupitermag: C++ library for field tracing in Jupiter's magnetosphere

GitHub: https://github.com/mattkjames7/libjupitermag.git

Code for obtaining magnetic field vectors and traces from within Jupiter's magnetosphere using various magnetic field models.

This is part of a community code project :

Magnetospheres of the Outer Planets Group Community Code

This module forms part of the JupiterMag package for Python.

3.6.1 Cloning and Building

This module requires a few submodules to be fetched, so the following command should clone everything:

```
git clone --recurse-sumodules https://github.com/mattkjames7/libjupitermag.git
```

This library requires `g++`, `make`, and `ld` (Linux) or `libtool` (Mac) in order to be compiled. On Windows, these tools can be provided by TDM-GCC.

To build in Linux and Mac, simply run

```
cd libjupitermag
make
#optionally install the library
sudo make install
```

where the installation defaults to `/usr/local` but can be changed using the `PREFIX` argument, e.g.:

```
sudo make install PREFIX=/usr
```

It can also be uninstalled:

```
make uninstall
```

Under Windows powershell/command line:

```
cd libjupitermag
compile.bat
```

or under Linux but building for Windows:

```
cd libjupitermag
make windows
```

After a successful build, a new library (`libjupitermag.so`) or DLL (`libjupitermag.dll`) should appear in the `lib/libjupitermag` directory.

3.6.2 Usage

The shared object library created by compiling this project should be accessible by various programming languages. This section mostly covers C++, other languages should be able to use the functions defined in the `extern "C"` section of the header file quite easily. For Python, it is straightforward to use `ctypes` to access this library, similarly, IDL could use the `CALL_EXTERNAL` function.

Linking to the library

Here is a very basic example of how to link to the code using C++ and print the version of the library:

```
/* test.cc */
#include <stdio.h>
#include <jupitermag.h>

int main() {
        /* simply print the version of the library */
        printf("libjupitermag version: $%d.%d.%d\n",
                        LIBJUPITERMAG_VERSION_MAJOR,
                        LIBJUPITERMAG_VERSION_MINOR,
                        LIBJUPITERMAG_VERSION_PATCH);
        return 0;
}
```

If the library was installed using `sudo make install`, then the library can be linked to during compiling time with:

```
g++ test.c -o test -ljupitermag
```

Otherwise, the header should be included using a relative or absolute path, e.g:

```
/* instead of this */
#include <jupitermag>
/* use something like this */
#include "include/jupitermag.h"
```

then compile, e.g.:

```
# absolute path
g++ test.cc -o test -L:/path/to/libjupitermag.so

# or relative path
g++ test.cc -o test -Wl,-rpath='$$ORIGIN/../lib/libjupitermag' -L ..lib/libjupitermag -ljupitermag
```

Calling Field Models

Internal field models can be called using the `InternalModel` object, e.g.:

```
#include <stdio.h>
#include <jupitermag.h>

int main () {
        /* create an instance of the object */
        InternalModel modelobj = InternalModel();

        /* set the model to use */
        modelobj.SetModel("jrm09");

        /* get the model vectors at some position */
        double x = 10.0, y = 0.0, z = 0.0;
        double Bx, By, Bz;
        modelobj.Field(x,y,z,&Bx,&By,&Bz);

        printf("B at [%f,%f,%f] = [%f,%f,%f]\n",x,y,z,Bx,By,Bz);
}
```

There are also simple functions for each of the models included in the library, see the table below.

Model	C String	Field Function	Reference
JRM33	`jrm33`	`jrm33Field`	Connerney et al., 2022
JRM09	`jrm09`	`jrm09Field`	Connerney et al., 2018
ISaAC	`isaac`	`isaacField`	Hess et al., 2017
VIPAL	`vipal`	`vipalField`	Hess et al., 2011
VIP4	`vip4`	`vip4Field`	Connerney 2007
VIT4	`vit4`	`vit4Field`	Connerney 2007
O4	`o4`	`o4Field`	Connerney 1981
O6	`o6`	`o6Field`	Connerney 2007
GSFC15evs	`gsfc15evs`	`gsfc15evsField`	Connerney 1981
GSFC15ev	`gsfc15ev`	`gsfc15evField`	Connerney 1981
GSFC13ev	`gsfc13ev`	`gsfc13evField`	Connerney 1981
Ulysses 17ev	`u17ev`	`u17evField`	Connerney 2007
SHA	`sha`	`shaField`	Connerney 2007
Voyager 1 17ev	`v117ev`	`v117evField`	Connerney 2007
JPL15ev	`jpl15ev`	`jpl15evField`	Connerney 1981
JPL15evs	`jpl15evs`	`jpl15evsField`	Connerney 1981
P11A	`p11a`	`p11aField`	
External Model			
Con 2020	`con2020`	`Con2020Field`	Connerney et al., 1981; Edwards et al., 2001; Connerney et al., 2020

Table 3.1: Internal and External Field Models

Field Tracing

Field tracing can be done using the `Trace` object, e.g.:

```
#include <stdio.h>
#include <jupitermag.h>
#include <vector>
```

```
int main () {
        /* set initial position to start trace from (this can be an array
                for multiple traces) */
        int n = 1;
        double x0 = 5.0;
        double y0 = 0.0;
        double z0 = 0.0;
        int nalpha = 1;
        double alpha = 0.0;

        printf("Create field function vector\n");
        /* store the function pointers for the components of the
        model to be included in the trace */
        std::vector<FieldFuncPtr> Funcs;

        /* internal model */
        Funcs.push_back(jrm09Field);

        /* external model */
        Funcs.push_back(Con2020Field);

        /* initialise the trace object */
        printf("Create Trace object\n");
        Trace T(Funcs);

        /* add the starting positions for the traces */
        printf("Add starting position\n");
        T.InputPos(n,&x0,&y0,&z0);

        /* configure the trace parameters, leaving this empty will
        use default values for things like minimum and maximum step size */
        printf("Set the trace parameters \n");
        T.SetTraceCFG();

        /* set up the alpha calculation - the angles (in degrees) of each
        polarization angle. This is generally used for ULF waves */
        printf("Initialize alpha\n");
        T.SetAlpha(nalpha,&alpha);

        /* Trace */
        printf("Trace\n");
        T.TraceField();

        /* trace distance, footprints, Rnorm */
        printf("Footprints etc...\n");
        T.CalculateTraceDist();
        T.CalculateTraceFP();
        T.CalculateTraceRnorm();

        /* calculate halpha for each of the polarization angles
           specified above*/
        printf("H_alpha\n");
        T.CalculateHalpha();
}
```

The above code traces along the magnetic field using the JRM09 internal and Con2020 external field models together. The trace coordinates and field vectors at each step can be obtained from the member variables `T.x_`, `T.y_`, `T.z_`, `T.Bx_`, `T.By_`, and `T.Bz_`, where each is a 2D array with the shape (`T.n_`,`T.MaxLen_`), where `T.n_` is the number of traces and `T.MaxLen_` is the maximum number of steps allowed in the trace. The number of steps taken in each trace is defined in the `T.nstep_` array.

3.7 con2020: Python implementation of Jupiter's magnetodisc model

3.8 con2020

Python implementation of the Connerney et al., 1981 and Connerney et al., 2020 Jovian magnetodisc model. This model provides the magnetic field due to a "washer-shaped" current near to Jupiter's magnetic equator. This model code uses either analytical equations from Edwards et al., 2001 or the numerical integration of the Connerney et al., 1981 equations to provide the magnetodisc field, depending upon proximity to the disc along z and the inner edge of the disc, r_0.

For the IDL implementation of this model, see: https://github.com/marissav06/con2020$_idl$

Or for Matlab: https://github.com/marissav06/con2020$_matlab$

A PDF documentation file is available here: con2020$_final_code_documentation_june9_2022.pdf$. It describes the Connerney current sheet model and general code development (equations used, numerical integration assumptions, accuracy testing, etc.). Details specific to the Python code are provided in this readme file.

These codes were developed by Fran Bagenal, Marty Brennan, Matt James, Gabby Provan, Marissa Vogt, and Rob Wilson, with thanks to Jack Connerney and Masafumi Imai. They are intended for use by the Juno science team and other members of the planetary magnetospheres community. Our contact information is in the documentation PDF file.

3.8.1 Installation

Install the module using `pip3`:

```
pip3 install --user con2020

#or if you have previously installed using this method
pip3 install --upgrade --user con2020
```

Or using this repo:

```
#clone the repo
git clone https://github.com/gabbyprovan/con2020
cd con2020

#EITHER create a wheel and install (X.X.X is the current version number)
python3 setup.py bdist_wheel
pip3 install --user dist/con2020-X.X.X-py3-none-any.whl

#or directly install using setup.py
python3 setup.py insall --user
```

3.8.2 Usage

To call the model, an object must be created first using `con2020.Model()`, where the default model parameters, model equations used or coordinate systems of input and output can be altered using keywords, e.g:

```
import con2020

#initialize a model object with default parameters
def_model = con2020.Model()

#initialize a model which uses spherical polar coordinates for input and output
sph_model = con2020.Model(CartesianIn=False,CartesianOut=False)

#initialize a model with custom parameters (longhand)
cust_model0 = con2020.Model(mu_i_div2__current_parameter_nT=150.0,
                                                    r0__inner_rj=9.5,
                                                d__cs_half_thickness_rj=3.1)

#equivalently, a custom parameter model (shorthand)
cust_model1 = con2020.Model(mu_i=150.0,r0=9.5,d=3.1)
```

Once a model object is initialized, the model field can be obtained by calling the member function `Field()` and supplying input coordinates as three scalars, or three arrays (all of which are in right-handed System III), e.g.:

```
#Example 1: the model at a single Cartesian position (all in Rj)
x = 5.0
y = 10.0
z = 6.0
Bcart = def_model.Field(x,y,z)
#Result:
Bxyz=[15.57977074, 36.88229249, 63.02051163] nT
#Calculated using the default con2020 model keywords and the hybrid approximation.

#Example 2: the model at an array of positions of spherical polar coordinates
r = np.array([10.0,20.0])                                    #radial distance in Rj
theta = np.array([30.0,35.0])*np.pi/180.0        #colatitude in radians
phi = np.array([90.0,95.0])*np.pi/180.0        #east longitude in radians
Bpol = sph_model.Field(r,theta,phi)
#Result:
Spherical polar Brtp =[63.32354453 ,31.15790459], [-21.01051861 , -6.86773727], [-3.61151705, -2.726260
Cartesian       Bxyz =[3.61151705, 1.6486016], [13.4661294,  12.43672946], [65.34505753, 29.46223351] n
#Calculated using the default con2020 model keywords and the hybrid approximation.
```

The output will be a `numpy.ndarray` with a shape `(n,3)`, where `n` is the number of input coordinates, `B[:,0]` corresponds to either `Bx` or `Br`; `B[:,1]` corresponds to `By` or `Btheta`; and `B[:,2]` corresponds to either `Bz` or `Bphi`. A full list of model keywords is shown below:

Keyword (long)	Keyword (short)	Default Value	Description
mu_i_div2__current_parameter_nT	mu_i	139.6*	Current sheet current density
i_rho__radial_current_MA	i_rho	16.7*	†Radial current intensity in M from Connerney et al 2020.
r0__inner_rj	r0	7.8	Inner edge of the current sheet
r1__outer_rj	r1	51.4	Outer edge of the current shee
d__cs_half_thickness_rj	d	3.6	Current sheet half thickness in
xt__cs_tilt_degs	xt	9.3	Tilt angle of the current sheet
xp__cs_rhs_azimuthal_angle_of_tilt_degs	xp	155.8	(Right-Handed) Longitude tow
equation_type		'hybrid'	Which method to use, can be: 'analytic' - use only the ana 'integral' - numerically inte 'hybrid' - a combination of a
error_check		True	Check errors on inputs the the
CartesianIn		True	If True (default) then the inpu
CartesianOut		True	If True the output magnetic fi
azfunc		'connerney'	Which model to use for the az
DeltaRho		1.0	Scale length over which smoot
DeltaZ		0.1	Scale length over which smoot
g		417659.3836476442	§Magnetic dipole parameter, n
w0_open		0.1	§Ratio of plasma to Jupiter's a
w0_om		0.35	§Ratio of plasma to Jupiter's a
thetamm		16.1	§Colatitude of the centre of th
dthetamm		0.5	§Colatitude range over which t
thetaoc		10.716	§Colatitude of the centre of th
dthetaoc		0.125	§Colatitude range of the open-

*Default current densities used here are averages provided in Connerney et al., 2020 (see Figure 6), but can vary from one pass to the next. Table 2 of Connerney et al., 2020 provides a list of both current densities for 23 out of the first 24 perijoves of Juno.

† This is only applicable for the Connerney et al., 2020 model for B_ϕ.

§ These parameters are used to configure the L-MIC model for B_ϕ.

The `con2020.Test()` function should produce figure 3.4:

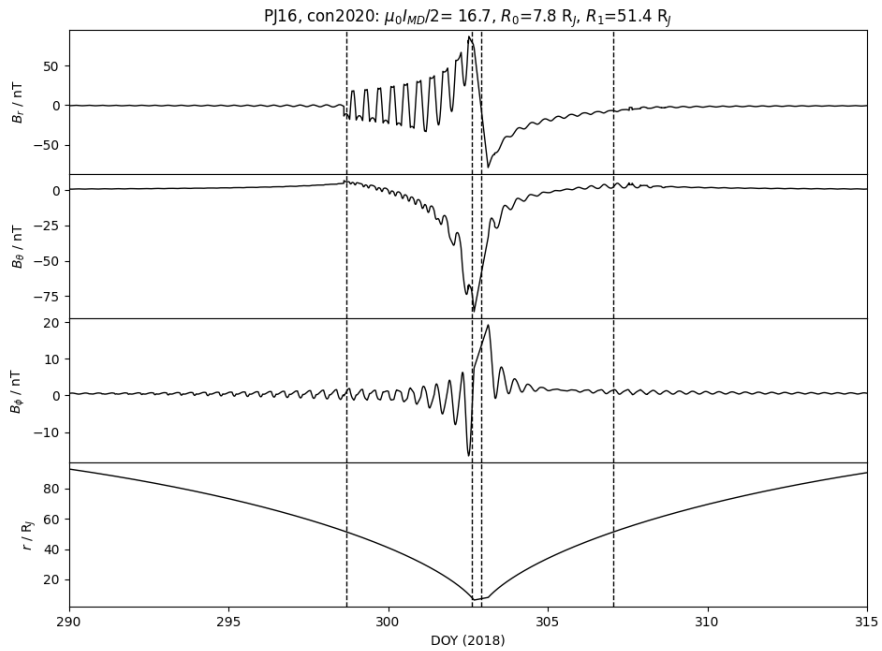

Figure 3.4: Test of `con2020` code on a Juno orbit.

3.9 `libcon2020`: C++ implementation of Jupiter's magnetodisc model

3.10 `jrm33`: The JRM33 model in Python

3.11 `jrm09`: The JRM09 model in Python

3.12 `vip4model`: The VIP4 model in Python

Chapter 4

Spacecraft Data

Chapter 5

Ground Data and Geomagnetic Indices

5.1 groundmag: Tools for processing and reading ground magnetometer data

5.2 SuperDARN: Simple SuperDARN fitacf reading code

https://github.com/mattkjames7/SuperDARN

The `SuperDARN` module is for reading and plotting SuperDARN fitacf files. It is a fairly simple tool, but use with caution because there may be some errors...

5.2.1 Installation

This package is not in the PyPI, so manual installation is necessary:

```
#clone the repo
git clone https://github.com/mattkjames7/SuperDARN
cd SuperDARN

#build a Python package
python3 setup.py bdist_wheel

#install it (replace 0.1.0 with whatever version is built)
pip3 install dist/SuperDARN-0.1.0-py3-none-any.whl --user
```

Once installed, the directory used to create the Python wheel file can be deleted. It can be uninstalled using `pip3 uninstall SuperDARN`.

Before running for the first time, a couple of environment variables need to be set up to tell the module where to look for fitacf files and to say where it is able to store some files:

```
#path to where FITACF files are stored
#(this one is specific to SPECTRE)
export FITACF_PATH=/data/sol-ionosphere/fitacf

#path to where this module can create some files
#(this should be a path where you have write access)
export SUPERDARN_PATH=/some/other/path/SuperDARN
```

This module will not currently run on Windows (as far as I am aware) because it requires the compilation of some C++ code which is not yet cross-platform.

Usage

In ipython, the first time this module is imported, it should attempt to download some files from the Radar Software Toolkit (RST) which help in calculating the coordinates of the fields of view of each radar. These files are created in the path defined be the `$SUPERDARN_PATH` variable.

Reading Data

There are a few functions within `SuperDARN.Data` which provide objects containing data:

```
import SuperDARN as sd

#get the data from a single cell (Radar,Date,ut,Beam,Gate)
cdata = sd.Data.GetCellData('han',20020321,[22.0,24.0],9,25)

#or a whole beam of data (Radar,Date,ut,Beam)
bdata = sd.Data.GetBeamData('han',[20020321,20020322],[22.0,24.0],7)

#data for the whole field of view (Radar,Date,ut)
#in this case, the output is a dict where each key is a beam number
#pointing to a recarray for each beam as produced by GetBeamData
rdata = sd.Data.GetRadarData('han',[20020321,20020322],[22.0,23.0])
```

In the above examples `bdata` and `cdata` are `numpy.recarray` objects, `rdata` is a `dict` object containing a `numpy.recarray` for each beam.

The fitacf data are stored in memory once loaded so that they don't need to be re-read every time the data are requested. To check how much memory is in use and to clear it:

```
#check memory usage in MB
sd.Data.MemUsage()

#clear memory
sd.Data.ClearData()
```

Plotting Data

There are a bunch of very simple plotting functions, e.g.:

```
import matplotlib.pyplot as plt

#create a figure
plt.figure(figsize=(8,11))

#plot the power along a beam
ax0 = sd.Plot.RTIBeam('han',[20020321,20020322],[23.0,1.0],9,[20,35],
                                        Param='P_l',ShowScatter=True,fig=plt,
                                        maps=[2,3,0,0],scale=[1.0,100.0],zlog=True,
                                        cmap='gnuplot')

#the velocity
ax1 = sd.Plot.RTIBeam('han',[20020321,20020322],[23.0,1.0],9,[20,35],Param='V',
                                        fig=plt,maps=[2,3,1,0])

#velocity along a range of latitudes at a ~constant longitude of 105
ax2 = sd.Plot.RTILat('han',[20020321,20020322],[23.0,1.0],105.0,Param='V',
                                        fig=plt,maps=[2,3,0,1])

#velocity along a range of longitudes at a ~constant latitude of ~70
ax3 = sd.Plot.RTILon('han',[20020321,20020322],[23.0,1.0],70.0,Param='V',
                                        fig=plt,maps=[2,3,1,1])

#some specific cells
beams = [1,5,7,2,8,4,9]
gates = [20,26,33,22,25,21,29]
ax4 = sd.Plot.RTI('han',[20020321,20020322],[23.0,1.0],beams,gates,
                                        Param='V',fig=plt,maps=[2,3,0,2])

#totally different FOV plot
```

```
ax5 = sd.Plot.FOVData('han',20020321,23.5,Param='V',fig=plt,maps=[2,3,1,2])


plt.tight_layout()
```

which should produce figure 5.1.

Fields of View

These may be wrong. Use with great caution.

The fields of view of each radar are stored as instances of the `SuperDARN.FOV.FOVObj` objects in memory and can be accessed using `GetFOV`, e.g.:

```
#get the object from memory
Date = 20020321
fov = sd.FOV.GetFOV('pyk',Date)

#use it to retrieve the FOV in mag coordinates
mlon,mlat = fov.GetFOV(Mag=True,Date=Date)

#plot it
ax = fov.PlotPolar(Background=[0.0,0.2,1.0],Continents=[0.0,1.0,0.2],
                                color='magenta',ShowBeams=False,ShowCells=False,
                                linewidth=2.0,Mag=True,Lon=True)

#add some cells
beams = [1,5,7,2,8,4,9]
gates = [20,26,33,22,25,21,29]
fov.PlotPolarCells(beams,gates,color='red',fig=ax,Mag=True,linewidth=2.0,Lon=True)
```

The above code should look like figure 5.2:

5.3 kpindex: Download the latext Kp indices

5.4 pyomnidata: Download the latext OMNI and solar flux data

5.5 smindex: Read the SuperMAG indices

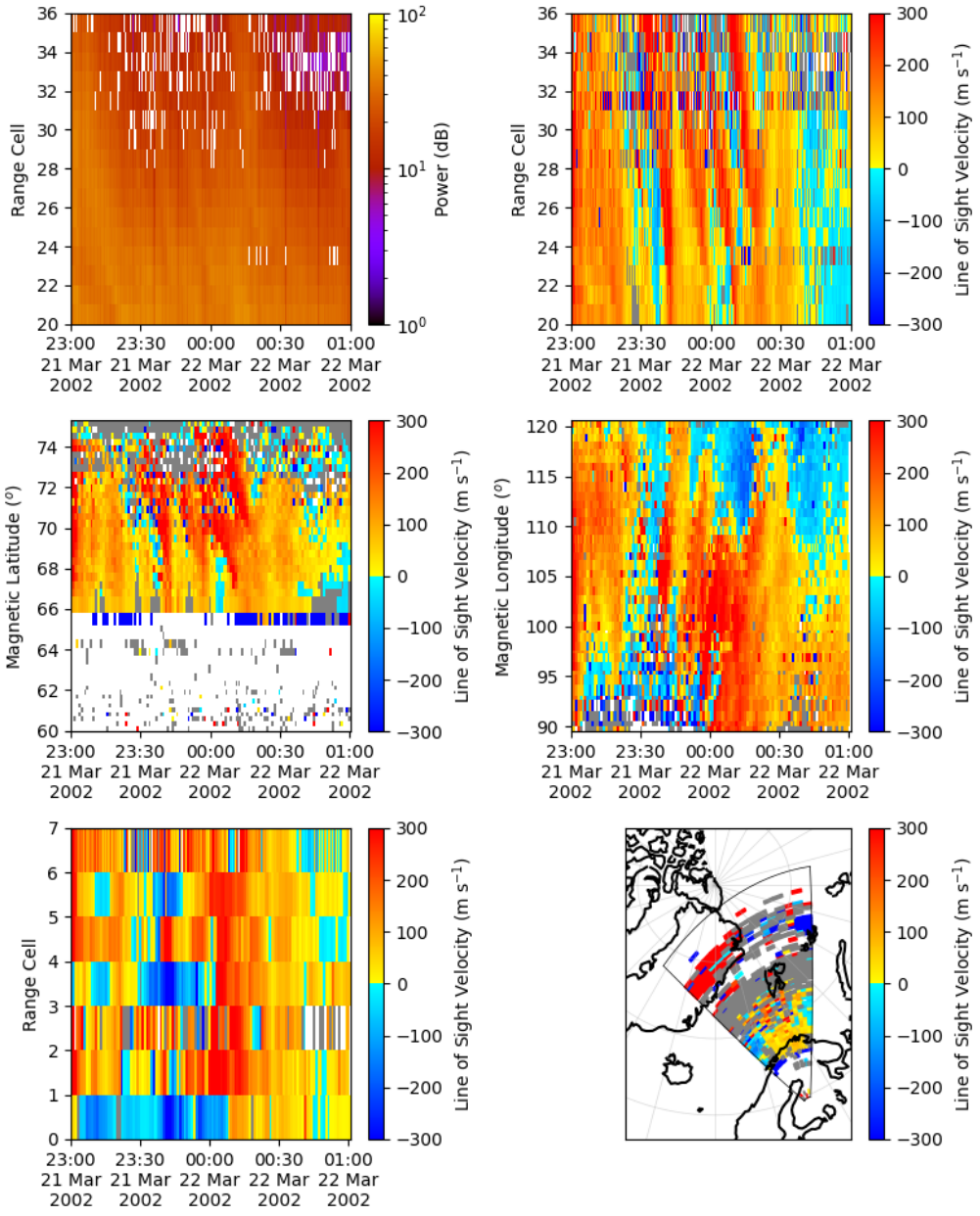

Figure 5.1: Top left: range time intensity (RTI) plot of backscatter power. Top right: RTI plot of line of sight velocity. Mid left: velocity along a line of cells in magnetic longitude. Mid right: velocity along a range of longitudes. Bottom left: velocity of specific range cells. Bottom right: velocity within the field of view plot.

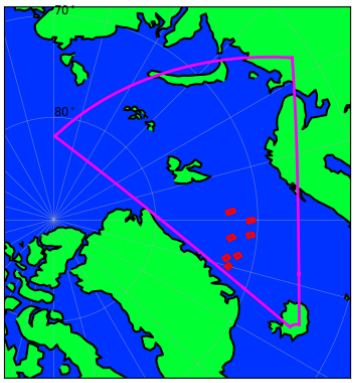

Figure 5.2: SuperDARN field of view plot with specific cells highlighted.

Chapter 6

Machine Learning

6.1 `NNClass`: Simple neural network classifier module

6.2 `NNFunction`: Train neural networks on arbitrary functions

Chapter 7

Other Tools

7.1 `wavespec`**: Spectral analysis tools**

7.2 `MHDWaveHarmonics`**: Tools for MHD waves**

7.3 `FieldTracing`**: Python field tracing code**

7.4 `DateTimeTools`**: Tools for dealing with dates and times**

7.5 `datetime`**: C++ library dealing for dates and times**

7.6 `PyFileIO`**: Tools for reading and writing files**

7.7 `RecarrayTools`**: Tools for manipulating** `numpy.recarrays`

7.8 `PBSJobExamples`**: Examples for submitting jobs to PBS**

7.9 `PlanetSpice`**: SPICE related code**

7.10 `ColorString`**: Change colour of strings in the terminal**

7.11 `cppembedbinary`**: Examples for embedding data into C++ code**

7.12 `libspline`**: C++ library for splines**

7.13 `linterp`**: C++ interpolation code**